PyQt4 入门指南

版本 1.0 翻译 hqwfreefly 日期 2011-5-5

声明：鄙人英文水平有限，若您对文中描述有异议，请以原文为准

分享知识　　传递快乐

1.PyQt4 工具包简介

1.1 关于本指南

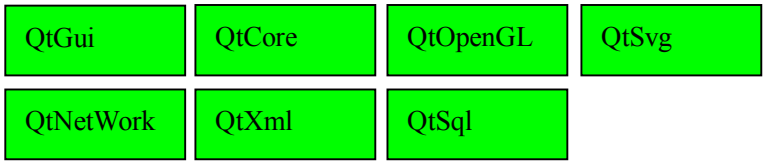
这是一个入门级的 PyQt 指南。其目的在于引导读者快速上手 PyQt4 工具包。该指南在 Linux 环境下创建并通过测试。

关于 PyQt

PyQt 是用来创建 GUI 应用程序的工具包。它是 Python 编程语言与已获得成功的 Qt 库的混合体。其中 Qt 库是这个星球上最强大的 GUI 库之一。PyQt 的官方网站是 http://www.riverbankcomputing.com/software/pyqt/intro 它由 Phil Thompson 创建。

PyQt 的实现被视作 Python 的一个模块。它由 300 多个类和接近 6000 个函数与方法构成。作为一个跨平台的工具包，PyQt 可以在所有主流的操作系统上运行（Unix、Windows、Mac）。PyQt 有两种许可，开发者可以在 GPL 和商业许可证之间做出选择。之前，PyQt 的 GPL 许可证只在 Unix 系统上可用，但在 PyQt4 之后，其 GPL 许可证适用于所有支持它的系统。

因为 PyQt 有大量的类，为便于管理，它们被划分到如下的几个模块中。



其中 QtCore 模块包含了核心的非 GUI 功能函数，用于以下方面：日期、文件和目录、数据结构、数据流、URL、MIME、线程和进程。QtGui 模块则包含了绘图组件以及与绘图相关的类，比如按钮、窗口、状态栏、工具栏、滑块、位图、颜色、字体等。QtNetWork 模块包含用于网络编程的类，用户可以用这些类实现 TCP/IP 和 UDP 的客户端或服务器。并且使用这些类会使网络编程更加容易、轻便。QtXml 包含用于处理 XML 文件的类，该模块提供了 SAX 和 DOM API 两种 XML 文件处理方式的实现。QtSvg 模块包含了用于显示 SVG（可缩放矢量图形，参考 http://zh.wikipedia.org/wiki/SVG）文件内容的类。QtOpenGL 模块用于渲染使用 OpenGL 库创建的 3D 或 2D 图形。并且它支持 Qt GUI 库和 OpenGL 库的无缝结合。QtSql 则库提供了用于操作数据库的类。

1.2 使用 PyQt4 创建入门程序

在本章的 PyQt4 指南中我们将学习一些基本的功能。我们讲解的速度会很慢，就像是在和一个孩子说话一样。对于一个孩子来说，他迈出的第一步是笨拙迟缓的。同样，对于一个编程新手来说，他接受新事物的过程也会比较的迟缓。但请谨记，没有愚蠢的人，只有懒人和人，并且懒人和人之间可以相互转换。

一个简单的示例

下面的示例代码非常简单，它只显示一个小窗口。然而，我们可以对窗口进行的操作却有很多，比如我们可以修改它的大小、最大化、最小化等。而这些操作却需要大量的代码，由于这些操作在很多程序中都需要用到，所以前人已经写好了这些操作的代码。我们没有必要一遍一遍的重新编写这些代码，因此这些代码对程序员来说是隐藏的。PyQt 是一个高度

抽象的工具包，因此，如果我们使用较底层的工具包来实现相同的功能，下面的示例代码就会增长很多。

```
#!/usr/bin/python
# simple.py
import sys
from PyQt4 import QtGui
app = QtGui.QApplication(sys.argv)
widget = QtGui.QWidget()
widget.resize(250, 150)
widget.setWindowTitle('simple')
widget.show()
sys.exit(app.exec_())
```

```
import sys
from PyQt4 import QtGui
```

这两句用来载入必须的模块。基本的 GUI 窗口部件在 QtGui 模块中。

app = QtGui.QApplication(sys.argv)

每一个 PyQt4 程序都需要有一个 application 对象，application 类包含在 QtGui 模块中。sys.argv 参数是一个命令行参数列表。Python 脚本可以从 shell 中执行，参数可以让我们选择启动脚本的方式。

widget = QtGui.QWidget()

QWidget 部件是 PyQt4 中所有用户界面类的父类。这里我们使用没有参数的默认构造函数，它没有继承其它类。我们称没有父类的 widget 为一个 window。

```
widget.resize(250, 150)
```

resize()方法可以改变窗口部件的大小，在这里我们将其设置为 250 像素宽，150 像素高。

```
widget.setWindowTitle('simple')
```

这句用来设置窗口部件的标题，该标题将在标题栏中显示。

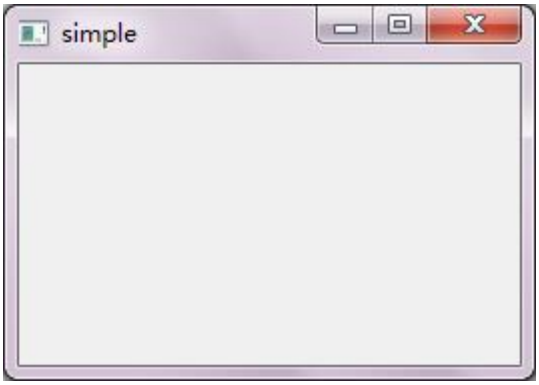
```
widget.show()
```

show()方法将窗口部件显示在屏幕上。

```
sys.exit(app.exec_())
```

最后我们进入该程序的主循环。事件处理从本行语句开始。主循环接受事件消息并将其分发给程序的各个部件。如果调用 exit()或主部件被销毁，主循环就会结束。使用 sys.exit()方法退出可以确保程序可以完整的结束，这种情况下系统的环境变量会记录程序是如何退出的。

也许你会疑惑，为什么 exec_()方法会有一个下划线。这是因为 exec 是 Python 的关键字，为避免冲突，PyQt 使用 exec_()替代。



截图：simple

1.3 程序图标

程序图标就是一个小图片，通常显示在程序标题栏的左上角。在以下的示例中，我们将学习如何在 PyQt 中使用程序图标，另外我们还将学习一些新的方法。

```
#!/usr/bin/python
# icon.py

import sys
from PyQt4 import QtGui

class Icon(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Icon')
        self.setWindowIcon(QtGui.QIcon('icons/web.png'))

app = QtGui.QApplication(sys.argv)
icon = Icon()
icon.show()
sys.exit(app.exec_())
```

上一个示例采用了面向过程的方法编写。 Python 语言同时支持面向过程和面向对象两种编程方法。PyQt 编程是面向对象的。

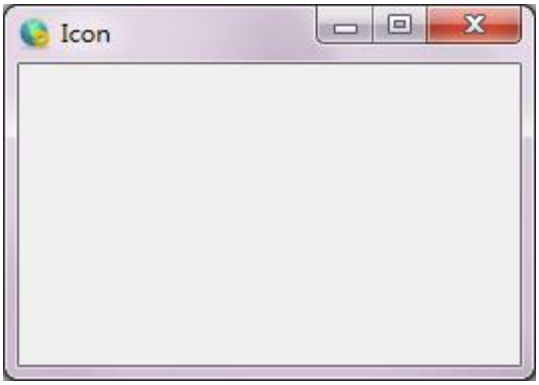
```
class Icon(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)
```

面向对象编程中最重要的是类、属性和方法。以上代码中，我们创建了一个名为 Icon 的新类，该类继承 QtGui.QWidget 类。因此我们必须调用两个构造函数——Icon 的构造函数

和继承类 QtGui.QWidget 类的构造函数。

```
self.setGeometry(300, 300, 250, 150)
self.setWindowTitle('Icon')
self.setWindowIcon(QtGui.QIcon('icons/web.png'))
```

setGeometry()方法完成两个功能——设置窗口在屏幕上的位置和设置窗口本身的大小。它的前两个参数是窗口在屏幕上的 x 和 y 坐标。后两个参数是窗口本身的宽和高。setWindowIcon()方法用来设置程序图标，它需要一个 QIcon 类型的对象作为参数。调用 QIcon 构造函数时，我们需要提供要显示的图标的路径（相对或绝对路径）。



截图：Icon

1.4 显示提示信息

我们可以为任何窗口部件设置一个气球提示。

```
#!/usr/bin/python
# tooltip.py

import sys
from PyQt4 import QtGui
from PyQt4 import QtCore

class Tooltip(QtGui.QWidget):
def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Tooltip')

        self.setToolTip('This is a <b>QWidget</b> widget')
        QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))
```

```
app = QtGui.QApplication(sys.argv)
tooltip = Tooltip()
tooltip.show()
sys.exit(app.exec_())
```

在本示例中，我们为一个 QWidget 类型的窗口部件设置工具提示。

```
self.setToolTip('This is a <b>QWidget</b> widget')
```

要创建工具提示，则需要调用 setToolTip()方法。该方法接受富文本格式的参数。

```
QtGui.QToolTip.setFont(QtGui.QFont('OldEnglish', 10))
```

因为默认的 QToolTip 字体看起来比较糟糕，我们可以通过上面的语句设置想要的字体和字体大小。

截图：tooltip

关闭窗口

一个显而易见的关闭窗口的方式是单击标题栏右上角的 X 标记。在接下来的示例中，我们将展示如何用代码来关闭程序，并简要介绍 Qt 的信号和槽机制。

下面是 QPushButton 的构造函数，我们将会在下面的示例中使用它。

```
QPushButton(string text, QWidget parent = None)
```

text 表示将显示在按钮上的文本。parent 是其对象，用于指定按钮显示在哪个部件中。在我们的示例中，parent 为是一个 QWidget 对象。

```
#!/usr/bin/python
# quitbutton.py

import sys
from PyQt4 import QtGui, QtCore

class QuitButton(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self, parent)

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('quitbutton')
```

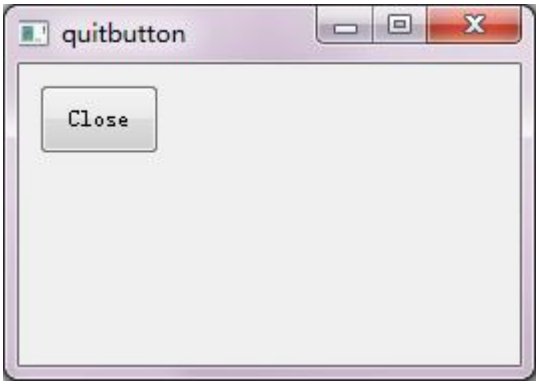
```
        quit = QtGui.QPushButton('Close', self)
        quit.setGeometry(10, 10, 60, 35)

        self.connect(quit, QtCore.SIGNAL('clicked()'), QtGui.qApp,
                QtCore.SLOT('quit()'))

app = QtGui.QApplication(sys.argv)
qb = QuitButton()
qb.show()
sys.exit(app.exec_())
```

```
quit = QtGui.QPushButton('Close', self)
quit.setGeometry(10, 10, 60, 35)
```

以上两句用来创建一个按钮并将其放在 QWidget 部件上，就像我们将 QWidget 部件放在屏幕上一样。

```
self.connect(quit, QtCore.SIGNAL('clicked()'), QtGui.qApp,QtCore.SLOT('quit()'))
```

PyQt4 的事件处理系统建立在信号-槽机制之上。如果我们单击 quit 按钮，那么信号 clicked()就会被触发，槽函数可以是 PyQt 自带的槽函数，也可以是任何 Python 可以调用的函数等。QtCore.QObject.connect()方法可以将信号和槽函数连接起来。在我们的示例中槽函数是 PyQt 中已定义的 quit()函数。通过 connect 方法就可以建立发送者（quit 按钮）和接受者（应用程序对象）之间的通信。



截图：quitbutton

消息窗口

默认情况下，如果我们单击了窗口标题栏上的 X 标记，窗口就会被关闭。但是有些时候我们想要改变这一默认行为。比如，我们正在编辑的文件内容发生了变化，这时若单击 X 标记关闭窗口，编辑器就应当弹出确认窗口。

```
#!/usr/bin/python
# messagebox.py

import sys
from PyQt4 import QtGui
```

```
class MessageBox(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)
                self.setGeometry(300, 300, 250, 150)
                self.setWindowTitle('message box')

        def closeEvent(self, event):
                reply = QtGui.QMessageBox.question(self, 'Message',
                                "Are you sure to quit?", QtGui.QMessageBox.Yes,
                                QtGui.QMessageBox.No)
                if reply == QtGui.QMessageBox.Yes:
                        event.accept()
                else:
                        event.ignore()

app = QtGui.QApplication(sys.argv)
qb = MessageBox()
qb.show()
sys.exit(app.exec_())
```

如果我们关闭 QWidget 窗口，QCloseEvent 事件就会被触发。要改变原有的 wdiget 行为阻止查窗口的关闭，我们就需要重新实现 closeEvent()方法。

```
reply = QtGui.QMessageBox.question(self, 'Message',
                "Are you sure to quit?", QtGui.QMessageBox.Yes,
                QtGui.QMessageBox.No)
```

通过上面的语句我们可以显示一个带有两个按钮（Yes/No）的消息窗口。第一个字符串参数'Message'在消息窗口的标题栏显示。第二个字符串参数以对话的形式显示在消息窗口中。返回的结果被保存在 reply 变量中。

```
if reply == QtGui.QMessageBox.Yes:
        event.accept()
else:
        event.ignore()
```

我们使用上面的 if 语句来判断用户选择的结果。如果用户选择了 Yes 按钮，那么关闭 widget 窗口并终止应用程序的动作会被允许执行。否则，关闭窗口的动作会被忽略。

截图：messagebox

将窗口放在屏幕中间

以下的脚本显示了将窗口放在屏幕的中间位置的方法。

```
#!/usr/bin/python
# center.py

import sys
from PyQt4 import QtGui

class Center(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)

                self.setWindowTitle('center')
                self.resize(250, 150)
                self.center()

        def center(self):
                screen = QtGui.QDesktopWidget().screenGeometry()
                size = self.geometry()
                self.move((screen.width() - size.width()) / 2,
                                (screen.height() - size.height()) / 2)


app = QtGui.QApplication(sys.argv)
qb = Center()
qb.show()
sys.exit(app.exec_())
```

---

self.resize()
该语句用来设置 QWidget 窗口的大小为 250 像素宽，150 像素高。
screen = QtGui.QDesktopWidget().screenGeometry()
该语句用来计算出显示器的分辨率（screen.width , screen.height）。
size = self.geometry()
该语句用来获取 QWidget 窗口的大小（size.width, size.height）。
self.move((screen.width() - size.width()) / 2, (screen.height() - size.height()) / 2)
该语句将窗口移动到屏幕的中间位置。

3.PyQt4 中的菜单和工具栏

主窗口

QMainWindow 类用来创建应用程序的主窗口。通过该类，我们可以创建一个包含状态栏、工具栏和菜单栏的经典应用程序框架。

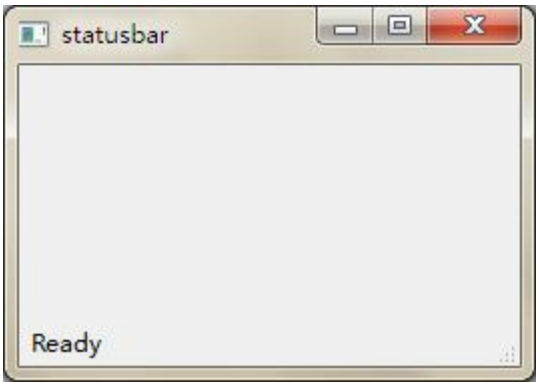
状态栏

状态栏是用来显示状态信息的串口部件。

```
#!/usr/bin/python
# statusbar.py

import sys
from PyQt4 import QtGui

class MainWindow(QtGui.QMainWindow):
        def __init__(self, parent = None):
                QtGui.QMainWindow.__init__(self)

                self.resize(250, 150)
                self.setWindowTitle('statusbar')
                self.statusBar().showMessage('Ready')

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
self.statusBar().showMessage('Ready')
```

使用 QApplication 类的 statusBar()方法创建状态栏。使用 showMessage()方法将信息显示在状态栏中。



截图：statusbar

菜单栏

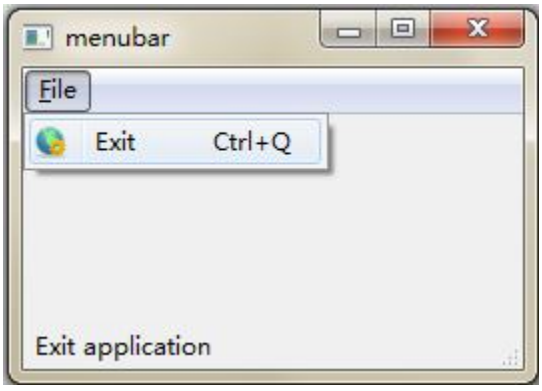
菜单栏是 GUI 程序最明显的组成部分。它由一组位于不同菜单中的命令组成。在控制台程序中，我们必须记住那些晦涩难懂的命令。但在 GUI 程序中，通过菜单栏我们将命令合理的放置在不同的菜单中来降低学习新应用程序的时间开销。

```
#!/usr/bin/python
```

```
# menubar.py

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
    def __init__(self, parent = None):
        QtGui.QMainWindow.__init__(self)

        self.resize(250, 150)
        self.setWindowTitle('menubar')

        exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
        exit.setShortcut('Ctrl+Q')
        exit.setStatusTip('Exit application')
        exit.connect(exit, QtCore.SIGNAL('triggered()'), QtGui.qApp,
                                                QtCore.SLOT('quit()'))

        self.statusBar()

        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
menubar = self.menuBar()
file = menubar.addMenu('&File')
file.addAction(exit)
```

首先我们使用 QMainWindow 类的 menuBar()方法创建一个菜单栏。然后使用 addMenu()方法添加一个菜单。最后我们把动作对象（这里是 exit）添加到 file 菜单中。



截图：menubar

工具栏

菜单对程序中的所有命令进行分组放置，而工具栏则提供了快速执行最常用命令的方法。

```
#!/usr/bin/python
# toolbar.py

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self, parent = None):
                QtGui.QMainWindow.__init__(self)
                self.resize(250, 150)
                self.setWindowTitle('toolbar')

                self.exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'),
                                'Exit', self)
                self.exit.setShortcut('Ctrl+Q')
                self.connect(self.exit, QtCore.SIGNAL('triggered()'),
                                QtGui.qApp, QtCore.SLOT('quit()'))
                self.toolbar = self.addToolBar('Exit')
                self.toolbar.addAction(self.exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

```
self.exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
self.exit.setShortcut('Ctrl+Q')
```

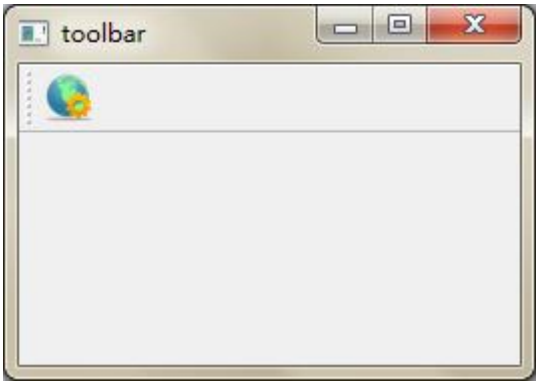
GUI 应用程序的行为是由命令来控制的，这些命令可以来自菜单、上下文菜单、工具栏或它们的快捷方式。PyQt 通过引入 actions 来简化编程难度，一个 action 对象可以拥有菜单、文本、图标、快捷方式、状态信息、“这是什么？”文本或工具提示等。在我们的示例程序中，我们定义了一个拥有图标、工具提示和快捷方式的 action 对象。

```
self.connect(self.exit, QtCore.SIGNAL('triggered()'), QtGui.qApp, QtCore.SLOT('quit()'))
```

该语句将 action 对象的 triggered()信号连接到预定义的 quit()槽函数。

```
self.toolbar = self.addToolBar('Exit')
```

该语句创建一个工具栏，然后使用语句 self.toolbar.addAction(self.exit)将 action 对象（这里是 exit）添加到该工具栏。



截图：toolbar

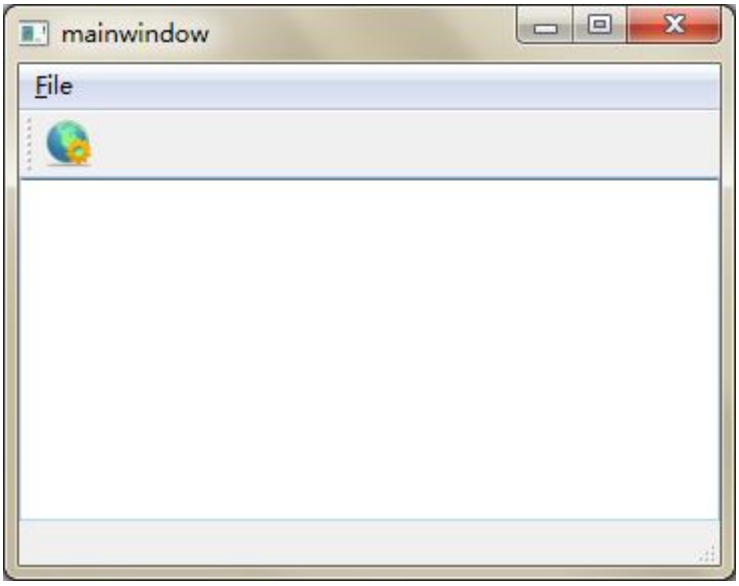
将它们聚合在一起

在本章的最后一个示例中，我们将创建一个菜单栏、一个工具栏和一个状态栏。我们还会创建一个中心部件。

```
#!/usr/bin/python
# mainwindow

import sys
from PyQt4 import QtGui, QtCore

class MainWindow(QtGui.QMainWindow):
        def __init__(self, parent = None):
                QtGui.QMainWindow.__init__(self)

                self.resize(350, 250)
                self.setWindowTitle('mainwindow')

                textEdit = QtGui.QTextEdit()
                self.setCentralWidget(textEdit)

                exit = QtGui.QAction(QtGui.QIcon('icons/exit.png'), 'Exit', self)
                exit.setShortcut('Ctrl+Q')
                exit.setStatusTip('Exit application')
                self.connect(exit, QtCore.SIGNAL('triggered()'), QtGui.qApp,
                                QtCore.SLOT('quit()'))

                self.statusBar()

                menubar = self.menuBar()
                file = menubar.addMenu('&File')
                file.addAction(exit)
```

```
        toolbar = self.addToolBar('Exit')
        toolbar.addAction(exit)

app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

在该示例中，我们创建了一个文本编辑部件，并将它设置为 QMainWindow 的中心部件。中心部件将占据所有的窗口剩余空间。



截图：mainwindow

4.PyQt4 中的布局管理器

布局管理器是编程中重要的一部分。所谓布局管理是指我们在窗口中安排部件位置的方法。布局管理有两种工作方式：绝对定位方式（absolute positioning）和布局类别方式（layout classes）。

绝对定位方式

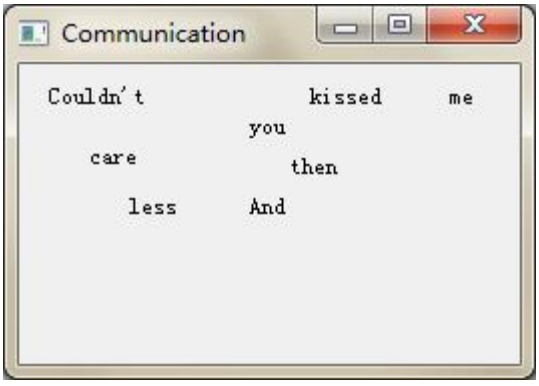
该方式下，程序员编程指定每一个部件的位置和尺寸像素。当使用绝对定位方式时，需要注意以下几点：

- 改变窗口大小时，窗口中部件的大小和位置不会随之改变。
- 在不同的平台上，应用程序可能会看起来不尽相同。
- 在应用程序中改变字体可能会导致布局混乱。
- 如果你打算改变窗口布局，你就必须得重新书写所有部件的布局，这一工作会非常乏味且耗时较多。

```
#!/usr/bin/python
# absolute.py
```

```
import sys
from PyQt4 import QtGui

class Absolute(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                self.setWindowTitle('Communication')
                label = QtGui.QLabel('Couldn\'t', self)
                label.move(15, 10)
                label = QtGui.QLabel('care', self)
                label.move(35, 40)
                label = QtGui.QLabel('less', self)
                label.move(55, 65)
                label = QtGui.QLabel('And', self)
                label.move(115, 65)
                label = QtGui.QLabel('then', self)
                label.move(135, 45)
                label = QtGui.QLabel('you', self)
                label.move(115, 25)
                label = QtGui.QLabel('kissed', self)
                label.move(145, 10)
                label = QtGui.QLabel('me', self)
                label.move(215, 10)

                self.resize(250, 150)

app = QtGui.QApplication(sys.argv)
qb = Absolute()
qb.show()
sys.exit(app.exec_())
```

在该示例中，我们简单是使用 move()方法来设置部件的位置。我们通过 x 和 y 坐标来指定 QLabel 部件的位置，坐标起点为左上角的顶点。x 坐标从左向右增长，y 坐标从上向下增长。



截图：absolute positioning

Box 布局

使用布局类别方式的布局管理器比绝对定位方式的布局管理器更加灵活实用。它是窗口部件的首先布局管理方式。最基本的布局类别是 QHBoxLayout 和 QVBoxLayout 布局管理方式，分别将窗口部件水平和垂直排列。

假设我们要将两个按钮放在窗口的右下角。为创建该布局，我们需要使用一个水平 Box 和一个垂直 Box，另外为了创建必须的空白空间，我们还需要添加一个伸缩间隔元素（stretch factor）。

```
#!/usr/bin/python
# boxlayout.py

import sys
from PyQt4 import QtGui
class Boxlayout(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                self.setWindowTitle('box layout')

                ok = QtGui.QPushButton('OK')
                cancel = QtGui.QPushButton('Cancel')

                hbox = QtGui.QHBoxLayout()
                hbox.addStretch(1)
                hbox.addWidget(ok)
                hbox.addWidget(cancel)

                vbox = QtGui.QVBoxLayout()
                vbox.addStretch(1)
                vbox.addLayout(hbox)

                self.setLayout(vbox)
                self.resize(300, 150)

app = QtGui.QApplication(sys.argv)
qb = Boxlayout()
qb.show()
sys.exit(app.exec_())
```

```
ok = QtGui.QPushButton('OK')
cancel = QtGui.QPushButton('Cancel')
```

以上两句用来创建两个按钮（OK 和 Cancel 按钮）。

```
hbox = QtGui.QHBoxLayout()
hbox.addStretch(1)
hbox.addWidget(ok)
hbox.addWidget(cancel)
```

以上四句用来创建一个水平 box 布局，然后加入一个伸缩间隔元素与两个按钮。

```
vbox = QtGui.QVBoxLayout()
vbox.addStretch(1)
vbox.addLayout(hbox)
```

为创建需要的布局，我们使用以上语句创建了一个垂直 box 布局并将水平 box 布局放入水平 box 布局中。

```
self.setLayout(vbox)
```

最后我们设置窗口的主布局。

截图：box layout

网格布局

最通用的布局类别是网格布局（QGridLayout）。该布局方式将窗口空间划分为许多行和列。要创建该布局方式，我们需要使用 QGridLayout 类。

```
#!/usr/bin/python
# gridlayout.py

import sys
from PyQt4 import QtGui

class GridLayout(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                self.setWindowTitle('grid layout')

                names = ['Cls', 'Bck', '', 'Close', '7', '8', '9', '/',
                                '4', '5', '6', '*', '1', '2', '3',
                                '-', '0', '.', '=', '+']
```

```
        grid = QtGui.QGridLayout()
        j = 0
        pos = [(0, 0), (0, 1), (0, 2), (0, 3),
                        (1, 0), (1, 1), (1, 2), (1, 3),
                        (2, 0), (2, 1), (2, 2), (2, 3),
                        (3, 0), (3, 1), (3, 2), (3, 3),
                        (4, 0), (4, 1), (4, 2), (4, 3)]
        for i in names:
                button = QtGui.QPushButton(i)
                if j == 2:
                        grid.addWidget(QtGui.QLabel(''), 0, 2)
                else:
                        grid.addWidget(button, pos[j][0], pos[j][1])
                j = j + 1

        self.setLayout(grid)

app = QtGui.QApplication(sys.argv)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

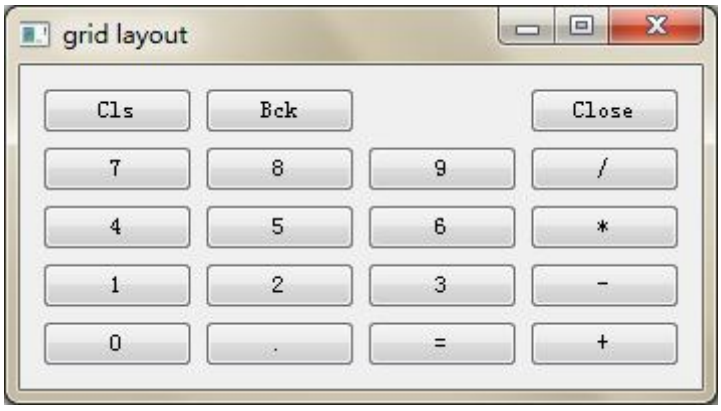
在这个示例中，我们创建一组按网格布局的按钮。为了填补 Bck 和 Close 按钮之间的空白，我们使用 QLabel 部件。

```
grid = QtGui.QGridLayout()
```

该语句创建了一个网格布局。

```
if j == 2:
        grid.addWidget(QtGui.QLabel(''), 0, 2)
else:
        grid.addWidget(button, pos[j][0], pos[j][1])
```

使用 addWidget()方法，我们将部件加入到网格布局中。addWidget()方法的参数依次为要加入到局部的部件，行号和列号。



截图：grid layout

部件在网格布局中可以跨越多行或多列。我们将下面的示例中演示该情况。

```
#!/usr/bin/python
# gridlayout2.py

import sys
from PyQt4 import QtGui

class GridLayout(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                self.setWindowTitle('grid layout')

                title = QtGui.QLabel('Title')
                authot = QtGui.QLabel('Author')
                review = QtGui.QLabel('Review')

                titleEdit = QtGui.QLineEdit()
                authorEdit = QtGui.QLineEdit()
                reviewEdit = QtGui.QLineEdit()

                grid = QtGui.QGridLayout()
                grid.setSpacing(10)

                grid.addWidget(title, 1, 0)
                grid.addWidget(titleEdit, 1, 1)

                grid.addWidget(authot, 2, 0)
                grid.addWidget(authorEdit, 2, 1)

                grid.addWidget(review, 3, 0)
                grid.addWidget(reviewEdit, 3, 1, 5, 1)

                self.setLayout(grid)
                self.resize(350, 300)

app = QtGui.QApplication(sys.argv)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

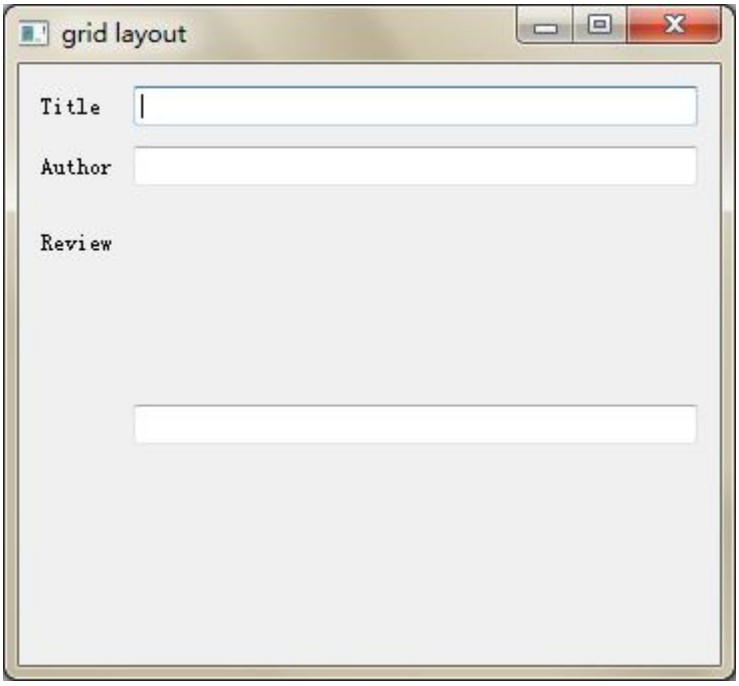
```
grid = QtGui.QGridLayout()
grid.setSpacing(10)
```

通过以上两句，我们创建了一个网格布局，并将该布局中的部件间隔（同行的横向间隔）

设为 10 个字距。

grid.addWidget(reviewEdit, 3, 1, 5, 1)

我们可以为加入网格布局的部件设置行列跨度，在上面的语句中，我们将 reviewEdit 部件的行跨度设置为 5，列跨度设置为 1。



截图：grid layout 2

5.PyQt4 的事件与信号

在本章的学习中，我们将介绍发生在应用程序中的事件和信号。

事件

事件（Events）是 GUI 程序中很重要的一部分。它由用户或系统产生。当我们调用程序的 exec_()方法时，程序就会进入主循环中。主循环捕获事件并将它们发送给相应的对象进行处理。为此，奇趣公司（Trolltech）引入了信号与槽机制。

信号与槽

当用户单击一个按钮，拖动一个滑块或进行其它动作时，相应的信号就会被发射。除此之外，信号还可以因为环境的变化而被发射。比如一个运行的时钟将会发射时间信号等。而所谓的槽则是一个方法，该方法将会响应它所连接的信号。在 Python 中，槽可以是任何可以被调用的对象。

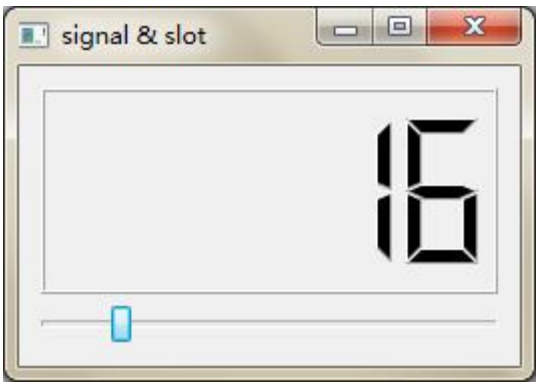
```
#!/usr/bin/python
# sigslot.py

import sys
from PyQt4 import QtGui, QtCore
```

```
class SigSlot(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)

                self.setWindowTitle('signal & slot')
                lcd = QtGui.QLCDNumber(self)
                slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)

                vbox = QtGui.QVBoxLayout()
                vbox.addWidget(lcd)
                vbox.addWidget(slider)

                self.setLayout(vbox)
                self.connect(slider, QtCore.SIGNAL('valueChanged(int)'), lcd,
                                QtCore.SLOT('display(int)'))
                self.resize(250, 150)

app = QtGui.QApplication(sys.argv)
qb = SigSlot()
qb.show()
sys.exit(app.exec_())
```

在这个示例中，我们创建了一个 LCD 显示器和一个滑块。通过拖动滑块我们可改变 LCD 显示器的数字。

self.connect(slider, QtCore.SIGNAL('valueChanged(int)'), lcd, QtCore.SLOT('display(int)'))

这里我们将滑块的 valueChanged()信号连接到 LCD 显示器的 display()槽函数上。连接方法 connect 有 4 个参数：信号发送者对象（这里是 slider 对象），要发射的信号（这里是 valueChanged 信号），信号的接收者对象（这里是 lcd 对象），对信号做出响应的槽函数（这里是 display 方法）。



截图：signal & slot

重写事件处理方法

PyQt 中的事件处理主要依赖重写事件处理函数来实现。

```
#!/usr/bin/python
# escape.py

import sys
from PyQt4 import QtGui, QtCore

class Escape(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                self.setWindowTitle('escape')
                self.resize(250, 150)
                self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
                                                QtCore.SLOT('close()'))

        def keyPressEvent(self, event):
                if event.key() == QtCore.Qt.Key_Escape:
                        self.close()

app = QtGui.QApplication(sys.argv)
qb = Escape()
qb.show()
sys.exit(app.exec_())
```

在上面的示例中，我们重新实现了 keyPressEvent()事件处理方法。

```
def keyPressEvent(self, event):
        if event.key() == QtCore.Qt.Key_Escape:
                self.close()
```

通过上面的方法，当我们按下 ESC 键时程序就会结束。

发射信号

继承自 QtCore.QObject 的对象可以均可以发射信号。如果我们单击一个按钮，那么一个 clicked()信号就会被触发。在接下来的示例中，我们将学习如何手动发射一个信号。

```
#!/usr/bin/python
# emit.py

import sys
from PyQt4 import QtGui, QtCore
class Emit(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)
```

```
                self.setWindowTitle('emit')
                self.resize(250, 150)
                self.connect(self, QtCore.SIGNAL('closeEmitApp()'),
                                        QtCore.SLOT('close()'))

        def mousePressEvent(self, event):
                self.emit(QtCore.SIGNAL('closeEmitApp()'))

app = QtGui.QApplication(sys.argv)
qb = Emit()
qb.show()
sys.exit(app.exec_())
```

在以上的示例中，我们创建了一个新的信号 closeEmitApp()，该信号在按下鼠标事件发生时被发射。

```
self.emit(QtCore.SIGNAL('closeEmitApp()'))
```

使用 PyQt 内建的 emit 函数发射信号 closeEmitApp()。

```
self.connect(self, QtCore.SIGNAL('closeEmitApp()'), QtCore.SLOT('close()'))
```

使用 connect 函数将手动创建的 closeEmitApp()信号和程序的 close()槽函数连接起来。这样在用户按下鼠标的任意键时，程序就会结束。

6.PyQt4 中的对话框

对话窗口和对话框是现代 GUI 应用程序必不可少的一部分。生活中“对话”被定义为发生在两人或更多人之间的会话。而在计算机世界，“对话”则是人与应用程序之间的“会话”。人机对话的形式有在输入框内键如内容，修改已有的数据，改变应用程序的设置等。对话框在人机交互中扮演着非常重要的角色。

从本质上说，只存在两种形式的对话框：预定义对话框和定制对话框。

预定义对话框

QInputDialog 输入对话框

QInputDialog 提供了一种获取用户单值数据的简洁形式。它接受的数据有字符串，数字和列表中的一项数据等。

```
#!/usr/bin/python
# inputdialog.py

import sys
from PyQt4 import QtGui, QtCore

class InputDialog(QtGui.QWidget):
        def __init__(self, parent = None):
```

```
        QtGui.QWidget.__init__(self)

        self.setGeometry(300, 300, 350, 80)
        self.setWindowTitle('InputDialog')
        self.button = QtGui.QPushButton('Dialog', self)
        self.button.setFocusPolicy(QtCore.Qt.NoFocus)
        self.button.move(20, 20)
        self.connect(self.button, QtCore.SIGNAL('clicked()'), self.showDialog)
        self.setFocus()

        self.label = QtGui.QLineEdit(self)
        self.label.move(130, 22)

    def showDialog(self):
        text, ok = QtGui.QInputDialog.getText(self, 'Input Dialog',
                                              'Enter your name:')
        if ok:
            self.label.setText(unicode(text))

app = QtGui.QApplication(sys.argv)
icon = InputDialog()
icon.show()
sys.exit(app.exec_())
```

本示例包含一个按钮和一个行编辑部件。单机按钮会弹出输入对话框，以获取用户输入的文本数据。该文本数据将会显示在行编辑部件中。

```
text, ok = QtGui.QInputDialog.getText(self, 'Input Dialog', 'Enter your name:')
```

该语句用来显示一个输入对话框。第一个参数'Input Dialog'是对话框的标题。第二个参数'Enter your name'将作为提示信息显示在对话框内。该对话框将返回用户输入的内容和一个布尔值，如果用户单击 OK 按钮确认输入，则返回的布尔值为 true，否则返回的布尔值为 false。

截图：Input Dialog

QColorDialog 颜色对话框

QcolorDialog 提供了用于选择颜色的对话框。

```
#!/usr/bin/python
# colordialog.py
```

```
import sys
from PyQt4 import QtGui, QtCore

class ColorDialog(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                color = QtGui.QColor(0, 0, 0)
                self.setGeometry(300, 300, 250, 180)
                self.setWindowTitle('ColorDialog')

                self.button = QtGui.QPushButton('Dialog', self)
                self.button.setFocusPolicy(QtCore.Qt.NoFocus)
                self.button.move(20, 20)

                self.connect(self.button, QtCore.SIGNAL('clicked()'), self.showDialog)
                self.setFocus()

                self.widget = QtGui.QWidget(self)
                self.widget.setStyleSheet('QWidget {background-color: %s}' %
                                        color.name())
                self.widget.setGeometry(130, 22, 100, 100)
        def showDialog(self):
                col = QtGui.QColorDialog.getColor()
                if col.isValid():
                        self.widget.setStyleSheet('QWidget {background-color: %s}' %
                                col.name())

app = QtGui.QApplication(sys.argv)
qb = ColorDialog()
qb.show()
sys.exit(app.exec_())
```

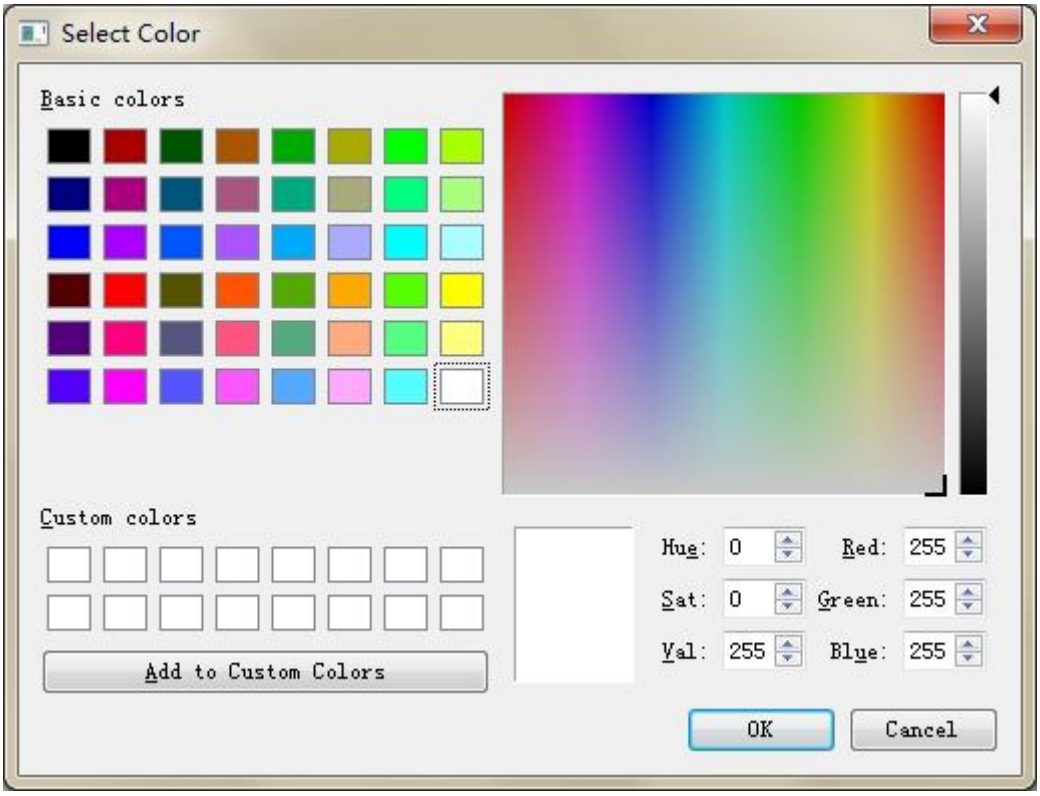
以上示例程序显示了一个按钮和一个 QWidget 部件，该 widget 部件的初始背景颜色为黑色。使用颜色对话框 QcolorDialog，我们可以改变 widget 部件的背景色。

```
col = QtGui.QColorDialog.getColor()
```

这一行语句用来弹出颜色对话框。

```
if col.isValid():
    self.widget.setStyleSheet('QWidget {background-color: %s}'  %  col.name())
```

以上的语句首先检测颜色时候可用。如果用户单击了颜色对话框的取消按钮，则对话框将不返回任何可用的颜色。如果颜色可用，我们就使用 stylesheets 设置 widget 部件的背景色。



截图：Color Dialog

QFontDialog 字体对话框

字体对话框用来一个用来这是字体的对话框部件。

```
#!/usr/bin/python
# fontdialog.py

import sys
from PyQt4 import QtGui, QtCore

class FontDialog(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)

                hbox = QtGui.QHBoxLayout()

                self.setGeometry(300, 300, 250, 110)
                self.setWindowTitle('FontDialog')

                button = QtGui.QPushButton('Dialog', self)
                button.setFocusPolicy(QtCore.Qt.NoFocus)
                button.move(20, 20)

                hbox.addWidget(button)
```

```
                self.connect(button, QtCore.SIGNAL('clicked()'), self.showDialog)

                self.label = QtGui.QLabel('Knowledge only matters', self)
                self.label.move(130, 20)

                hbox.addWidget(self.label, 1)
                self.setLayout(hbox)

        def showDialog(self):
                font, ok = QtGui.QFontDialog.getFont()
                if ok:
                        self.label.setFont(font)

app = QtGui.QApplication(sys.argv)
cd = FontDialog()
cd.show()
sys.exit(app.exec_())
```

在本示例中，我们在主界面中显示了一个按钮和一个标签。单机按钮后，用户可在弹出字体对话框中选择字体来修改标签中的字体样式。

```
hbox.addWidget(self.label, 1)
```

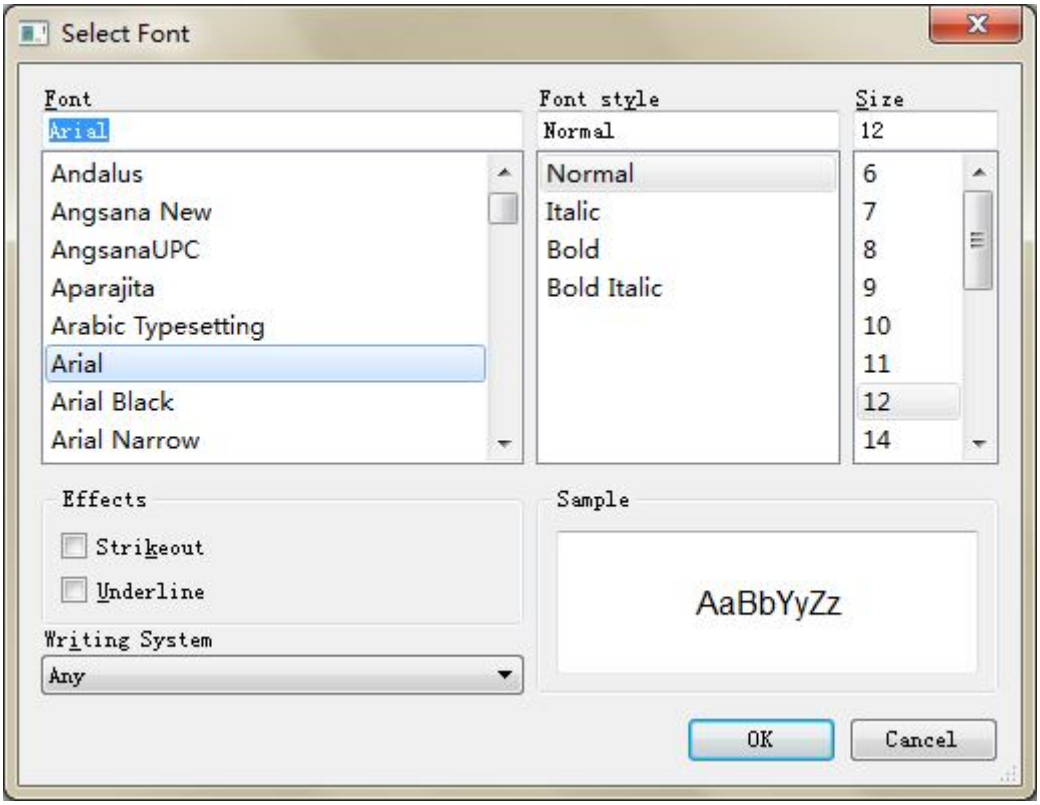
该语句将 labe 标签加入到 hbox 布局中，并通过第二个参数 1 设置 label 的大小是可变的。该设置是必须的，因为在用户选择不同的字体时，label 标签中的字体可能会变大，若不进行该设置，标签中的内容就可能不会被全部显示。

```
font, ok = QtGui.QFontDialog.getFont()
```

该语句将弹出字体对话框。

```
if ok:
    self.label.setFont(font)
```

在用户选择了字体并单击 OK 按钮后，使用标签对象的 setFont 方法设置标签内容的字体。



截图：Font Dialog

QFileDialog 文件对话框

文件对话框允许用户选择文件或文件夹，被选择的文件可进行读或写操作。

```
#!/usr/bin/python
# openfiledialog.py

import sys
from PyQt4 import QtGui, QtCore

class OpenFile(QtGui.QMainWindow):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)

                self.setGeometry(300, 300, 350, 300)
                self.setWindowTitle('OpenFile')
                self.textEdit = QtGui.QTextEdit()
                self.setCentralWidget(self.textEdit)
                self.statusBar()
                self.setFocus()

                exit = QtGui.QAction(QtGui.QIcon('icons/open.png'), 'Open', self)
                exit.setShortcut('Ctrl+O')
                exit.setStatusTip('Open new file')
```

```
                self.connect(exit, QtCore.SIGNAL('triggered()'), self.showDialog)

                menubar = self.menuBar()
                file = menubar.addMenu('&File')
                file.addAction(exit)

        def showDialog(self):
                filename = QtGui.QFileDialog.getOpenFileName(self, 'Open file', './')
                file = open(filename)
                data = file.read()
                self.textEdit.setText(data)

app = QtGui.QApplication(sys.argv)
cd = OpenFile()
cd.show()
sys.exit(app.exec_())
```

我们在本示例程序中显示了一个菜单栏，一个状态栏和一个被设置为中心部件的文本编辑器。其中状态栏的状态信息只有在用户想要打开文件时才会显示。单击菜单栏中的 Open 选项将弹出文件对话框供用户选择文件。被选择的文件内容将被显示在文本编辑器部件中。

```
class OpenFile(QtGui.QMainWindow):
        ...
                self.textEdit = QtGui.QTextEdit()
                self.setCentralWidget(self.textEdit)
```

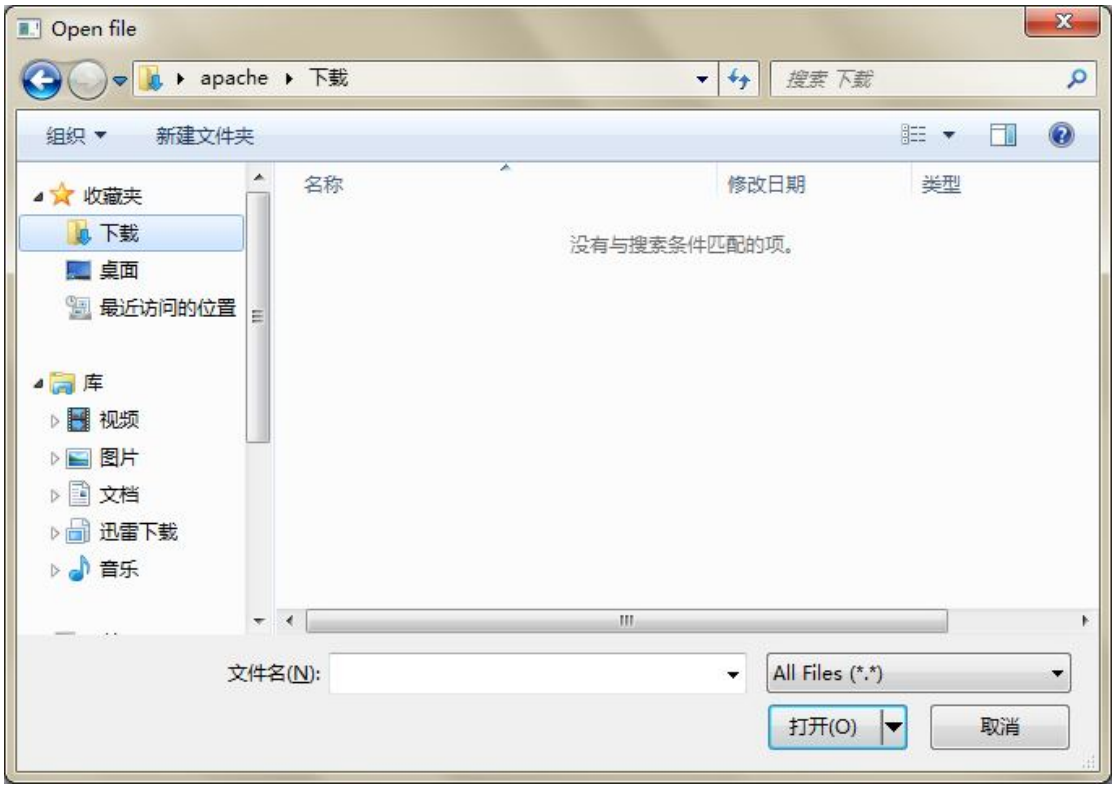
本示例程序是基于 QMainWindow 窗口部件的，因为我们需要将文本编辑器设置为中心部件（QWidget 部件类没有提供 setCentralWidget 方法）。无须依赖布局管理器，QMainWindow 即可轻松完成设置中心部件的工作（使用 setCentralWidget 方法）。

```
filename = QtGui.QFileDialog.getOpenFileName(self, 'Open file', './')
```

该语句将弹出文件对话框。getOpenFileName()方法的第一个字符串参数'Open File'将显示在弹出对话框的标题栏。第二个字符串参数用来指定对话框的工作目录。默认情况下文件过滤器被设置为不过滤任何文件（所有工作目录中的文件/文件夹都会被显示）。

```
ile = open(filename)
data = file.read()
self.textEdit.setText(data)
```

以上三行语句将读取被选择的文件并将其内容显示在文本编辑器中。



截图：File Dialog

7.PyQt4 中的部件

部件是构建应用程序的基础元素。PyQt4 工具包拥有大量的种类繁多的部件。比如：按钮，单选框，滑块，列表框等任何程序员在完成其工作时需要的部件。本本章的学习中，我们将介绍一些有用的部件。

QCheckBox 单选框

单选框具有两个状态：被选中或未被选中。它看起来像一个附件了标签的方框。当用户选择或取消选择时，单选框就会发射一个 stateChanged()信号。

```
#!/usr/bin/python
# checkbox.py
import sys
from PyQt4 import QtGui, QtCore
class CheckBox(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)

                self.setGeometry(300, 300, 250, 150)
                self.setWindowTitle('Checkbox')

                self.cb = QtGui.QCheckBox('Show title', self)
                self.cb.setFocusPolicy(QtCore.Qt.NoFocus)
```

```
                self.cb.move(10, 10)
                self.cb.toggle()
                self.connect(self.cb,QtCore.SIGNAL('stateChanged(int)'),
                                        self.changeTitle)
        def changeTitle(self, value):
                if self.cb.isChecked():
                        self.setWindowTitle('Checkbox')
                else:
                        self.setWindowTitle('Unchecked')

app = QtGui.QApplication(sys.argv)
w = CheckBox()
w.show()
sys.exit(app.exec_())
```

在本示例中，我们创建了一个用来改变窗口标题的单选框。

```
self.cb = QtGui.QCheckBox('Show title', self)
```

该语句用来创建一个标签信息为'Show title'的单选框。

```
self.connect(self.cb, QtCore.SIGNAL('stateChanged()'), self.changeTitle)
```

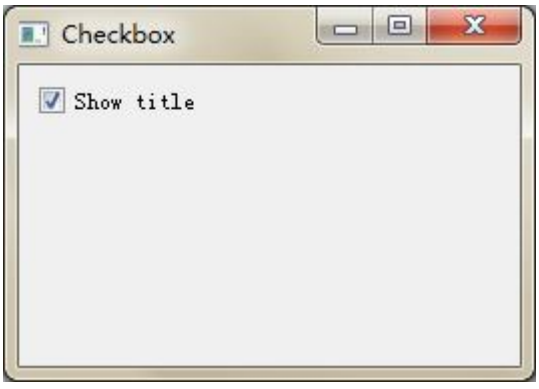
这里我们将用户定义的 changeTitle()函数与单选框的 stateChanged()信号连接起来。自定义的 changeTitle()函数将重置窗口的标题。

```
self.cb.setFocusPolicy(QtCore.Qt.NoFocus)
```

默认情况下单选框接受聚焦，被聚焦的表现形式为单选框的标签被一个薄薄的矩形所覆盖。这个矩形看起来回有些太过严肃，所以我们使用以上代码行将单选框的聚焦样式修改为 Qt.NoFocus 的无聚焦样式。

```
self.cb.toggle()
```

初始状态下我们设置了窗口的标题，因此我们需要使用以上代码行将单选框选上。在默认情况下，单选框是未被选中的。



截图：QCheckBox

ToggleButton 开关按钮

PyQt4 没有开关按钮部件。但是我们可以使用在特殊状态下的 QPushButton 部件来创建开关按钮。而所谓的开关按钮就是一个具有按下和未按下两种状态的普通按钮。用户可以通

过单击按钮来切换其开或关的状态。在一些情形下，这个特性会非常好用。

```
#!/usr/bin/python
# togglebutton.py
import sys
from PyQt4 import QtGui, QtCore
class ToggleButton(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)
                self.color = QtGui.QColor(0, 0, 0)
                self.setGeometry(300, 300, 280, 170)
                self.setWindowTitle('ToggleButton')
                self.red = QtGui.QPushButton('Red', self)
                self.red.setCheckable(True)
                self.red.move(10, 10)
                self.connect(self.red, QtCore.SIGNAL('clicked()'),
                                self.setRed)
                self.green = QtGui.QPushButton('Green', self)
                self.green.setCheckable(True)
                self.green.move(10, 60)
                self.connect(self.green, QtCore.SIGNAL('clicked()'),
                                self.setGreen)
                self.blue = QtGui.QPushButton('Blue', self)
                self.blue.setCheckable(True)
                self.blue.move(10, 110)
                self.connect(self.blue, QtCore.SIGNAL('clicked()'),
                                self.setBlue)
                self.square = QtGui.QWidget(self)
                self.square.setGeometry(150, 20, 100, 100)
                self.square.setStyleSheet('Qwidget {background-color: %s}' %
                                self.color.name())
                QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('cleanlooks'))

        def setRed(self):
                if self.red.isChecked():
                        self.color.setRed(255)
                else:
                        self.color.setRed(0)
                self.square.setStyleSheet('QWidget {background-color: %s}' %
                                self.color.name())
        def setGreen(self):
                if self.green.isChecked():
                        self.color.setGreen(255)
                else:
                        self.color.setGreen(0)
```

```
                self.square.setStyleSheet('QWidget {background-color: %s}' %
                        self.color.name())
        def setBlue(self):
                if self.blue.isChecked():
                        self.color.setBlue(255)
                else:
                        self.color.setBlue(0)
                self.square.setStyleSheet('QWidget {background-color: %s}' %
                        self.color.name())

app = QtGui.QApplication(sys.argv)
w = ToggleButton()
w.show()
sys.exit(app.exec_())
```

在这个例子中，我们创建了三个开关按钮和一个 QWidget 部件，并将 QWidget 部件的背景颜色设置为黑色（译注：笔者发现初始背景色实际上是白色）。用户通过开关按钮从红、绿、蓝选择出 QWidget 部件的背景颜色组合。若开关按钮被按下，则其对应的颜色即被选中。

```
self.color = QtGui.QColor(0, 0, 0)
```

这行语句用来设置初始颜色，红绿蓝三种颜色值均为 0 时的颜色为黑色。

```
self.red = QtGui.QPushButton('Red', self)
self.red.setCheckable(True)
```

通过创建一个 QPushButton 并将其设置为可被选择的，即得到我们想要的开关按钮。

```
self.connect(self.red, QtCore.SIGNAL('clicked()'), self.setRed)
```

我们将 red 开关按钮的 clicked()信号和自定义的 setRed()方法连接起来。

```
QtGui.QApplication.setStyle(QtGui.QStyleFactory.create('cleanlooks'))
```

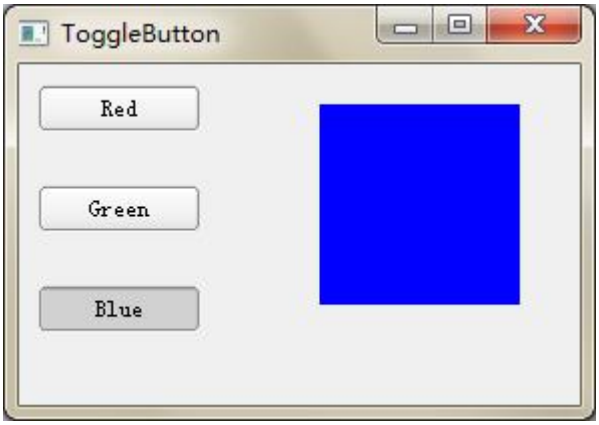
该行语句用来将应用程序的外观样式设置为 cleanlooks 的。之所以这样做是因为 Linux 系统下的默认样式存在一个小的设计缺陷，该缺陷使用户无法快速的分辨出开关按钮的两种状态。而采用 cleanlooks 样式外观的表现会好些。

```
if self.red.isChecked():
        self.color.setRed(255)
else:
        self.color.setRed(0)
```

我们使用 if 语句来判断开关按钮的状态并设置对应的颜色值。

```
self.square.setStyleSheet('Qwidget {background-color: %s}'  %  self.color.name())
```

使用 setStyleSheet()方法改变 QWidget 部件的背景色。



截图：ToggleButton

QSlider 滑块、QLabel 标签

滑块部件由一个简单的操控杆构成，用户可以通过向前或向后滑动滑块来选择数据。这种选择数据的方式对一些特殊的任务来说比单纯的提供一个数据或使用 spin box 调整数据大小的方式要自然与友好的多。而标签部件则用来显示文本或图片。

在以下的示例中，我们将在窗口中显示一个滑块和一个标签。这次我们将在标签部件中显示图片，并使用滑块来控制其显示内容。

```
#!/usr/bin/python
# slider-label.py
import sys
from PyQt4 import QtGui, QtCore

class SliderLabel(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self, parent)
                self.setGeometry(300, 300, 250, 150)
                self.setWindowTitle('SliderLabel')
                self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
                self.slider.setFocusPolicy(QtCore.Qt.NoFocus)
                self.slider.setGeometry(30, 40, 100, 30)
                self.connect(self.slider, QtCore.SIGNAL('valueChanged(int)'),
                                self.changeValue)
                self.label = QtGui.QLabel(self)
                self.label.setPixmap(QtGui.QPixmap('icons/mute.png'))
                self.label.setGeometry(160, 40, 80, 30)
        def changeValue(self, value):
                pos = self.slider.value()
                if pos == 0:
                        self.label.setPixmap(QtGui.QPixmap('icons/mute.png'))
                elif 0 < pos <= 30:
                        self.label.setPixmap(QtGui.QPixmap('icons/min.png'))
                elif 30 < pos < 80:
```

```
                self.label.setPixmap(QtGui.QPixmap('icons/med.png'))
        else:
                self.label.setPixmap(QtGui.QPixmap('icons/max.png'))

app = QtGui.QApplication(sys.argv)
w = SliderLabel()
w.show()
sys.exit(app.exec_())
```

在这个示例中我们模拟一个音量控制的场景，通过拖动滑块来改变标签部件中的图片显示。

```
self.slider = QtGui.QSlider(QtCore.Qt.Horizontal, self)
```

通过该语句我们创建了一个水平滑块部件。

```
self.label = QtGui.QLabel(self)
self.label.setPixmap(QtGui.QPixmap('icons/mute.png'))
```

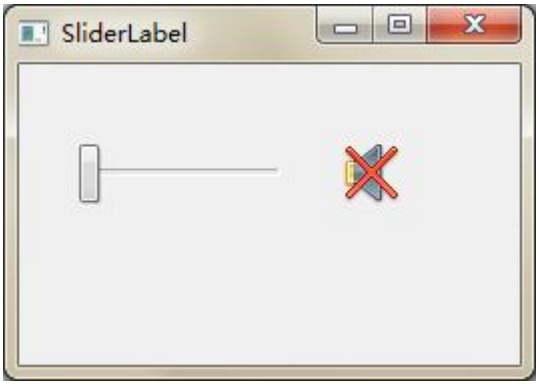
以上两行语句用来创建一个标签部件并将 mute.png 加入到该部件中显示。

```
self.connect(self.slider, QtCore.SIGNAL('valueChanged(int)'), self.changeValue)
```

这行语句将滑块的 valueChanged()信号与自定义的 changeValue()方法相连接。

```
pos = self.slider.value()
```

语句用来获取当前的滑块位置。



截图：Slider and Label

QProgressBar 进度条

当我们在处理一个耗时较长的任务时，可能就会用到进度条部件。因为使用进度条可以形象告诉用户当前的任务正在进行中。PyQt4 工具包提供了水平和垂直两种类型的进度条部件。我们可以设置进度条的最大和最小值，默认的最大和最小值分别为 0 和 99。

```
#!/usr/bin/python
# progressbar.py

import sys
from PyQt4 import QtGui, QtCore
class ProgressBar(QtGui.QWidget):
        def __init__(self, parent = None):
                QtGui.QWidget.__init__(self)
```

```
                self.setGeometry(300, 300, 250, 150)
                self.setWindowTitle('ProgressBar')

                self.pbar = QtGui.QProgressBar(self)
                self.pbar.setGeometry(30, 40, 200, 25)

                self.button = QtGui.QPushButton('Start', self)
                self.button.setFocusPolicy(QtCore.Qt.NoFocus)
                self.button.move(40, 80)

                self.connect(self.button, QtCore.SIGNAL('clicked()'), self.onStart)
                self.timer = QtCore.QBasicTimer()
                self.step =    0

        def timerEvent(self, event):
                if self.step >= 100:
                        self.timer.stop()
                        return
                self.step = self.step +1
                self.pbar.setValue(self.step)

        def onStart(self):
                if self.timer.isActive():
                        self.timer.stop()
                        self.button.setText('Start')
                else:
                        self.timer.start(100, self)
                        self.button.setText('Stop')

app = QtGui.QApplication(sys.argv)
icon = ProgressBar()
icon.show()
sys.exit(app.exec_())
```

在这个示例中，我们创建了一个水平的进度条和一个按钮。按钮用来启动或终止进度。

`self.pbar = QtGui.QProgressBar(self)`

使用该构造器来创建一个进度条。

`self.timer = QtCore.QBasicTimer()`

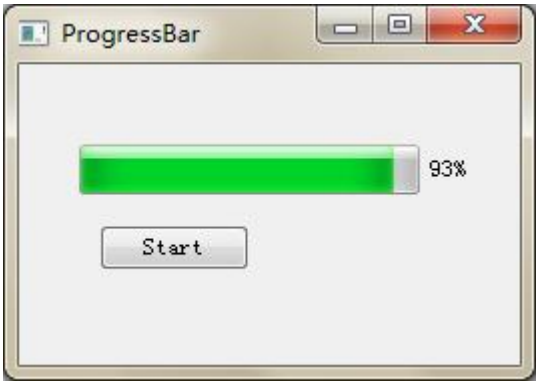
创建一个定时器对象。

`self.timer.start(100, self)`

要激活该进度条，我们需要使用定时器的 start()方法启动定时器。该方法的第一个参数为超时时间。第二个参数表示当前超时时间到了以后定时器触发超时事件的接收对象。

```
def timerEvent(self, event):
        if self.step >= 100:
            self.timer.stop()
            return
        self.step = self.step +1
        self.pbar.setValue(self.step)
```

每一个 QObject 对象或其子对象都有一个 QObject.timerEvent 方法。在本实例中，为了响应定时器的超时事件，我们需要使用上面的代码重写进度条的 timerEvent 方法。



截图：ProgressBar

QCalendarWidget 日历部件

QCalendarWidget 类提供了以月为单位的日历部件。该部件允许用户以一种简单而直接的方式选择日期。

```
#!/usr/bin/python
# calendar.py
import sys
from PyQt4 import QtGui, QtCore
class Calendar(QtGui.QWidget):
    def __init__(self, parent = None):
        QtGui.QWidget.__init__(self)
        self.setGeometry(300, 300, 350, 300)
        self.setWindowTitle('Calendar')
        self.cal = QtGui.QCalendarWidget(self)
        self.cal.setGridVisible(True)
        self.connect(self.cal, QtCore.SIGNAL('selectionChanged()'),
                     self.showDate)
        self.label = QtGui.QLabel(self)
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.cal)
        vbox.addWidget(self.label)
```

```
                self.setLayout(vbox)
        def showDate(self):
                date = self.cal.selectedDate()
                self.label.setText(str(date.toPyDate()))

app = QtGui.QApplication(sys.argv)
w = Calendar()
w.show()
sys.exit(app.exec_())
```

该示例窗口拥有一个日历部件和一个标签部件。当前被选中的日期被显示在标签部件中。

使用 self.cal = QtGui.QCalendarWidget(self)语句创建一个日历对象。

self.connect(self.cal, QtCore.SIGNAL('selectionChanged()'), self.showDate)语句将日历对象的 selectionChanged()信号和自定义的 showDate()方法相连接。这样在我们选定一个日期后，日历对象的 selectionChanged()消息就会被触发，而与之相连的 showDate()方法就会被调用。

```
def showDate(self):
        date = self.cal.selectedDate()
        self.label.setText(str(date.toPyDate()))
```

在 showDate 方法中，我们使用 selectedDate 方法获取被选择的日期。然后将其转换为字符串并显示在标签部件中。

截图：Calendar Widget